Practice Midterm 5 Solutions

This page contains solutions to Practice Midterm 5.

Question 1: C++ and ADTs (Co-occurrence Map)

This task was mostly about implementing a given specification and demonstrating that you know how to correctly use ADTs, including a nested collection type.

Solution 1

This solution uses nested loops to process each pair of strings within each sentence. Two sets are used to track strings we have already seen – one for strings from the outer loop and one for strings from the inner loop. A common error here is to use only one set (the one in the inner loop) while neglecting the need to avoid duplicates in the outer loop *or* using that set to overzealously cut strings from the outer loop.

There are many possible subtle variations on this solution.

```
Map<string, Map<string, int>> coOccurrenceMap(Vector<string>& sentences) {
  Map<string, Map<string, int>> map;

  for (string sentence : sentences) {
    Vector<string> words = stringSplit(sentence, " ");
    Set<string> seen1;

    for (int i = 0; i < words.size(); i++) {
      if (!seen1.contains(words[i])) {
        seen1.add(words[i]);
        Set<string> seen2;

        for (int j = 0; j < words.size(); j++) {
          if (words[i] != words[j] && !seen2.contains(words[j])) {
            seen2.add(words[j]);
            map[words[i]][words[j]]++;
          }
        }
      }
    }
  }

  return map;
}
```

Solution 2

This solution dumps all the strings from a sentence into a set before processing them with nested loops. This solution is more elegant and frees us from the tedium, complexity, and potential pitfalls associated with the way duplicates are tracked in the previous solution. This highlights the advantage of thinking in abstractions and relying on existing tools to solve problems for us wherever possible!

```
Map<string, Map<string, int>> coOccurrenceMap(Vector<string>& sentences) {
  Map<string, Map<string, int>> map;

  for (string sentence : sentences) {
    Vector<string> words = stringSplit(sentence, " ");
    Set<string> uniqueWords;

    for (string word : words) {
      uniqueWords.add(word);
    }

    for (string s1 : uniqueWords) {
      for (string s2 : uniqueWords) {
        if (s1 != s2) {
          map[s1][s2]++;
        }
      }
    }
  }

  return map;
}
```

Question 2: Runtime Analysis (Big-O)

This question asked you to give best- and worst-case Big-O runtimes and focused on tracing through functions that manipulated a stack.

stack contents when "Target found!" gets printed

- original (note that 80 is **not** in the stack):

```
| 92 | <- top
+----+
| 17 |
+----+
| 48 |
+----+
| 53 | <- bottom
+----+
```

- sideDish:

```
| 61 | <- top
+----+
| 33 | <- bottom
+----+
```

worst case for `seekAndDestroy()`

- Worst-Case Runtime: O(n)
- Stack Drawing: must contain five integers; should not contain 15, even if it's at the bottom of the stack

best case for `seekAndDestroy()`

- Best-Case Runtime: O(1)

- Stack Queue: must contain five integers; top must be 15; doesn't matter what the other integers are

`removeAllInstances()`

- All *n* values equal to *target*: O(n)
- No values equal to *target*: O(n)
- *k* top values not equal to *target*, bottom *m* values equal: O(km)

Question 3: Recursion (digitSum)

This recursive problem involved a small math puzzle and was shorter (and perhaps more manageable) than recursion questions in previous quarters to offset the time required for the other questions on this quarter's exam.

```
// Assumes non-negative n.
int digitSum(int n) {
  if (n == 0) {
    return 0;
  }

  // n % 10 extracts ones digit
  // n / 10 (int division) excises ones digit from value passed recursively
  return (n % 10) + digitSum(n / 10);
}
```

Question 4: Recursive Backtracking (countBaskets)

There are many ways to approach this problem, but the restrictions on the exam caused the structure of most solutions we saw to mirror that of one of the following two solutions.

Solution 1

```
int countBaskets(Vector<fruitT>& fruits, int minFruit, int maxFruit, int
minTypes) {
  int total = 0;
  if (maxFruit < 0 || minTypes - fruits.size() > 0) {
    return 0;
  }

  if (fruits.size() == 0) {
    // Note that the minTypes <= 0 check is unnecessary here
    // because of how the base case is constructed above.
    if (minFruit <= 0 && maxFruit >=0 && minTypes <= 0) {
      total++;
    }

    return total;
  }

  fruitT thisOne = fruits.remove(fruits.size() - 1);

  // without this fruit
  total += countBaskets(fruits, minFruit, maxFruit, minTypes);

  // with this fruit
  for (int i = 1; i <= thisOne.count; i++) {
    total += countBaskets(fruits, minFruit - i, maxFruit - i, minTypes - 1);
  }

  fruits.add(thisOne);
  return total;
}
```

Solution 2

```
int countBasketsHelper(Vector<fruitT>& fruits, int minFruit, int maxFruit,
int minTypes, int numFruit, int numTypes) {
  int total = 0;
  if (numFruit > maxFruit || fruits.size() + numTypes < minTypes) {
    return 0;
  }

  if (fruits.size() == 0) {
    if (numFruit >= minFruit && numFruit <= maxFruit && numTypes >=
minTypes) {
      total++;
    }

    return total;
  }

  fruitT thisOne = fruits.remove(fruits.size() - 1);

  // without this fruit
  total += countBasketsHelper(fruits, minFruit, maxFruit, minTypes,
numFruit, numTypes);

  // with this fruit
  for (int i = 1; i <= thisOne.count; i++) {
    total += countBasketsHelper(fruits, minFruit, maxFruit, minTypes,
numFruit + i, numTypes + 1);
  }

  fruits.add(thisOne);
  return total;
}

int countBaskets(Vector<fruitT>& fruits, int minFruit, int maxFruit, int
minTypes) {
  return countBasketsHelper(fruits, minFruit, maxFruit, minTypes, 0, 0);
}
```